ハードウェアによる同期処理機構に関する研究

平成18年度

三重大学大学院工学研究科
博士前期課程 情報工学専攻

伊賀崇幸

修士論文

題目

ハードウェアによる同期処理機構に関する研究

指導教員

近藤 利夫 教授

2007年

三重大学大学院 工学研究科
博士前期課程 情報工学専攻
計算機アーキテクチャ研究室

伊賀 崇幸 (405M501)

内容梗概

近年，マルチプロセッサ・システムが普及し，プログラムを分割して並列に実行する並列処理手法が広く用いられるようになっている．中でもプログラムを粒度の細かいスレッド等の処理単位で，大多数に分割する中・細粒度並列処理は，実行するマルチプロセッサ・システムの形態に強く依存しない手法として注目されている．しかしながらこの手法では，プログラムを細かく分割する事によるタスク数の増加により，タスク・スケジューリングやコンテキスト・スイッチ，同期処理等に起因するオーバーヘッドが増大し，大幅な性能低下を招く危険性がある．

この問題を低減する手法として，従来よりハードウェアでスケジューリングを支援するSSH(Scheduling Support Hardware)，およびハードウェアでコンテキスト・スイッチ処理を支援するCSS(Context switch Support System)が提案されている．これらの手法を用いる事で，スケジューリング及びコンテキスト・スイッチに起因するオーバーヘッドが低減され，高速化が実現された．しかしながら，同期処理によるオーバーヘッドが依然問題となっており，同期命令を多数含むプログラムでは充分な性能向上が得られないという問題があった．

そこで本研究では，同期処理をハードウェアで支援するS3(Synchronization Support System)を提案し，マルチプロセッサ・システムに組込むことで，同期処理に起因するオーバーヘッドの増加を抑え更なる高速化を目指す．

Abstract

Today, multiprocessor systems have spread and Parallelism on the system are widely used in every usage. Particularly, Middle-fine grain parallelism, dividing program into a large number of small threads, is paid attention to. Because it hardly relies on multiprocessor system environment. However, the overhead of task scheduling, context switching and synchronization process becomes large compared to the execution time of threads and it often makes performance down greatly.

In order to reduce these overheads, the Scheduling Support Hardware architecture(SSH) and Context switch Support System(CSS) are proposed. SSH accelerates the performance of the OS with hardware by scheduling threads concurrently with the thread execution on the CPU. CSS saves the registers the former thread used and fetches the registers used in next thread in parallel with the thread execution on the CPU. By using them, the overhead is reduced. However, the overhead of the synchronization process still causes performance down. This paper proposes the Synchronization Support System(S3) which supports synchronization process by snooping shared memory bus in parallel with the thread execution on CPU, and adds S3 to the multiprocessor system with SSH and CSS and evaluates them.

目次

図目次

表目次

1　はじめに

近年，マルチプロセッサ・システムが普及し，プログラムをDOループやサブルーチン，関数といった粒度の粗い単位で分割して，並列に実行する粗粒度並列処理が広く用いられるようになっている．しかし，粗粒度並列処理ではタスクの粒度が粗いため，均等な負荷分散が難しく，プロセッサ台数が異なるマルチプロセッサ・システムにおいて，常に性能を最大限引き出せるようなプログラムを記述するのは極めて困難である．

この問題を解決する方法の1つとして存在するのが，中・細粒度並列処理である．中・細粒度並列処理は，粗粒度並列処理に対してプログラムを非常に小さく分割する．また，プロセッサ台数と比較しても圧倒的に多い数のスレッド集合に分割する．そして，データフローマシンのように実行条件の整ったスレッドから動的にアイドルプロセッサに割り当てて実行することで，プロセッサ台数に強く依存しない並列処理を実現出来る．

しかし，このような中・細粒度並列処理を従来のOSを用いて実行するには問題がある．何故なら，扱うタスク数が増える事により，スケジューリングやコンテキスト・スイッチ，同期処理の回数が増加する．そのため，これらに起因するオーバーヘッドが増大し，中・細粒度の並列性を

有効に利用出来ないばかりでなく，大幅な性能低下を招く危険性があるからである．つまり中・細粒度並列処理を有効に利用するには，効率的なスケジューリング，コンテキスト・スイッチ，そして同期処理の実現が不可欠となる．

これらの問題を低減するための手法として，従来よりOSの機能の一部であるスレッドのスケジューリングや，CPU資源の割り当て／解放の機能をハードウェアで実行する事で細粒度な並列性を有効利用し，高速なスケジューリングの実現を目指すスケジューリング支援ハードウェア (Scheduling Support Hardware; SSH) を用いたマルチプロセッサ・アーキテクチャが提案されている [1][2][3]．SSHはスケジューリング専用ハードウェアを導入することで，CPU上のスレッド実行とスケジューリングを並列に行い，スケジューリングに要する時間を隠蔽している．また同様に，コンテキスト・スイッチに伴うレジスタの退避／復帰処理をハードウェア化し，高速なコンテキスト・スイッチの実現をするためのコンテキスト・スイッチ支援システム (Context switch Support System; CSS) を用いたアーキテクチャが提案されている．SSHとCSSにより，スケジューリングやコンテキスト・スイッチに起因するオーバーヘッドによる性能低下を抑え，高速化が実現されている．しかしながら，同期処理に起因する

オーバーヘッドが依然問題となっている．そこで本研究では同期処理に着目し，同期処理を支援するシステムS3(Synchronization Support System)を提案し，中・細粒度並列処理の更なる高速化を目指す．

以降，本論文では次のように構成される．まず，次章では研究の背景となる中・細粒度並列処理の概要とその問題点を説明し，研究の目的を述べる．3章では，関連研究として本研究と同様にハードウェアで同期処理を行う手法を挙げ，本研究との違いについて述べる．続く4章では，SSHとCSSについて述べ，5章では，提案する同期処理支援システム(S3)の詳細なアーキテクチャ及び動作について述べる．そして6章にて，S3の性能評価について述べ，最後に7章で結論と今後の展望について述べる．

2 研究背景

2.1 中・細粒度並列処理

近年，マルチプロセッサ・システムが普及し，プログラムを並列に実行するための技術である並列処理手法に関する研究が広く行われている[4, 5, 6, 7, 8, 9, 10]．この並列処理手法のうち，プログラムをDOループやサブルーチン，関数といった粗い単位に分割して，並列に実行する手法を粗粒度並列処理と呼ぶ．これに対して，CPU数より圧倒的大多数の

細かいスレッド等に分割して，並列に実行する手法を中・細粒度並列処理と呼ぶ．

並列処理手法を用いた処理で現在の主流となっているのは，粗粒度並列処理である．しかしながら粗粒度並列処理では，分割したタスクの粒度が粗いため均等な負荷分散が難しく，プロセッサ台数やネットワーク形態等が異なる様々なマルチプロセッサ・システムにおいて，性能を常に最大限引き出せるようなプログラムを記述する事は，極めて困難である．そこで，注目されているのが中・細粒度並列処理である．

中・細粒度並列処理ではプログラムを細かく，そしてCPU数よりも圧倒的大多数のスレッドに分割するため，均等な負荷分散が容易になり，実行するマルチプロセッサ・システムに強く依存しない性能が得られるとして期待されている．しかしながら問題点も存在しており，中・細粒度並列処理の利点を充分に活用出来ていないのが現状である．その問題を引き起こす要因となる，代表的な処理を以下に挙げる．

- タスク・スケジューリング
- コンテキスト・スイッチ
- スレッド間同期

以降，基本的な並列処理に関する事項をまとめた後，上記の処理について簡単な説明と共に，中・細粒度並列処理におけるそれぞれの問題点を説明する．

2.2 スレッドについて

ここで，各処理とその問題点を述べる前に，スレッドについて簡単に説明する．スレッドとは，プログラムの実行単位であるプロセス内に1つ以上存在する並列実行単位である．プログラムが起動されると，プロセスが生成される．その際，記憶領域やディスク資源がプロセス毎に割り当てられる．これに対してスレッドは，プロセスとして実行中のプログラム内で生成される．そのため，スレッド生成時にスレッドに割り当てられる記憶領域等の資源は，そのプロセスに割り当てられた資源の一部から割り当てられる．また，スレッドはCPU割り当ての単位であり，実行可能状態になると，スケジューリング対象となる．

スレッドを用いる利点として，スレッド間の通信コストの低さがある．プロセスでは，プロセス毎に独立した記憶領域を持っている．そのため，プロセス間での通信コストは大きくなる．それに対してスレッドでは，同じプロセス内の他のスレッドと記憶領域を共有しているため，プロセス

間通信より高速になる．

以上のような利点により，特に粒度の小さい処理を扱う並列処理技術では，スレッドが広く用いられている．

2.3 タスク・スケジューリング

タスク・スケジューリングとは，分割した処理の実行順序や，実行するCPUの割り当て等を，その処理の優先度やスケジューリング・ポリシに従って決定する処理の事である．並列処理では逐次処理と異なり，分割した処理の実行するタイミングを厳密に管理する必要がある．これは，分割した処理の実行するタイミングによって，最終的な結果を誤ってしまう危険性が存在するためである．そのため，CPU上での処理が切り替わる場合や，分割した処理が終了した場合等には再度タスク・スケジューリングを行う必要がある．つまりタスク・スケジューリングは，プログラムの開始から終了までの間に，複数回実行される．

中・細粒度並列処理では，分割数が粗粒度並列処理より大幅に多くなる．そのため，スケジューリング対象が増え，スケジューリング回数が分割数に応じて大幅に増加する．また，分割した1つの処理の開始から終了までに要する時間が短くなり，1回のスケジューリング時間が1つの

処理に対して相対的に長くなる．その結果，スケジューリングに要するオーバーヘッドが増大し，大幅な性能低下を招き，問題となっている．

2.4 コンテキスト・スイッチ

コンテキスト・スイッチとは，CPU 上のタスクを切り替える処理の事である．CPU 上で実行していたタスクが終了した場合や，何らかの理由で処理を続行出来なくなった等で，他のタスクへと実行権限を譲る場合にコンテキスト・スイッチが行われ，CPU 上でのタスクが切り替わる．具体的には，それまで実行していたタスクが使用していたレジスタの内容をメモリへと退避し，次に実行するタスクが必要とするレジスタの内容をメモリから復帰する処理である．また，現在普及している多くの CPU では，ユーザが利用出来るレジスタ数が，1 本あたり 32bit 又は 64bit のレジスタが32本であるのが一般的である．つまり，コンテキスト・スイッチ処理で単純に全てのレジスタを退避／復帰するには，それぞれ32回ずつ，合計64回ものメモリ・アクセスが必要となる．

これがオーバーヘッドとなり，性能低下の原因となる危険性がある．特に中・細粒度並列処理では分割数が非常に多いため，コンテキスト・スイッチ回数も増えて，メモリ・アクセスが頻発する．その結果，メモリ・

アクセス・レイテンシに起因するオーバーヘッドが，大幅な性能低下を招く危険性があり問題となっている．

2.5 スレッド間同期

スレッド間同期とは，複数のスレッドが共有資源に同時にアクセスすることで発生する競合的破壊を防ぐための排他制御や，スレッド間でプログラムの足並みを合わせるためのバリア同期のことである．以下に，それぞれについて詳細に説明する．

2.5.1 排他制御

排他制御とは，共有資源に対し複数のスレッドから同時にアクセスが発生した場合に，その内の1つのスレッドにのみ独占的に資源を利用させるための同期手法である．

一般的なソフトウェアでは，ロック変数の獲得／解放という手法を用いて排他制御を実現している．ロック変数とは，ただ1つのスレッドにのみロック獲得され，その間，他のスレッドはロック獲得出来ない．ロックを獲得したスレッドは，ロックを獲得している間は独占的に共有資源にアクセス出来る．そして，ロックを獲得していたスレッドが共有資源に対する処理を終えてロックを解放すると，そのロックの解放を待ってい

た他のスレッドの内，ただ1つのスレッドにのみ再びロック獲得される．このような流れで排他制御が実現されている．

ここで，排他制御を用いないと発生する問題の例を挙げる．2CPU 上で実行中の2つのスレッドがそれぞれカウンタを1度インクリメントする事で，インクリメントしたスレッド数をカウントする場合を考える．初期条件として，カウンタの値を0とする．2つのスレッドがカウンタをそれぞれ1回ずつインクリメントするので，最終的にカウンタが2となるのが正しい結果である．また，この際のカウンタ操作は大きく分けて以下の3ステップ必要とする．

1. カウンタのアドレスに格納されている値を読み出す．

2. 読み出した値をインクリメントする．

3. カウンタのアドレスにインクリメントした新しい値を書き込む．

それぞれのステップについて，(1) を「Read」，(2) を「Increment」，(3) を「Write」として，処理の流れを図 2.1 に示す．図 2.1 左側の (a) は排他制御を用いない場合である．図 2.1 から分かるように，排他制御を用いない場合，誤った結果を得ている．次に排他制御を行う場合を考える．(a) で問題なのは，上記 (1) から (3) の処理がアトミックに行われていなかっ

たことに起因する．そこで，(1) から (3) の処理が動作を行う前にロック変数による排他制御を行い，アトミック性の保証を行う．具体的な実行例を図 2.1 右側の (b) に示す．(b) では上記の 1〜3 のステップの他に，ロック獲得成功の「Lock : Yes」と，ロック獲得失敗の「Lock : No」，そしてロック解放の「Unlock」というステップが加わる．ロック獲得はロック変数「Lock」の値が「Unlocked」であれば成功し，「Locked」であれば失敗する．「Counter」はカウンタの値を表す．排他制御を用いている (b) では (1) から (3) の動作がアトミックに行われるため，正しい結果を得られる．

このように，並列処理を行う場合の共有資源の利用は厳密に行わなければならず，その際に排他制御が必須となる．

しかしながら，図 2.1 から分かる通り排他制御を用いる手法では，処理の本質的な部分であるカウンタのインクリメント処理の他に，ロックの獲得／解放という処理が追加される．そのため，(a) と比較して (b) の方は処理時間が増加している．また，(b) の CPU2 上で実行しているスレッドに関しては，「Lock」の値が「Unlocked」になるまでロックの獲得は成功しないにも関わらず，ロックの獲得を何度も試みていて，CPU 資源を浪費している．これがオーバーヘッドとなり，大幅な性能低下を招く危

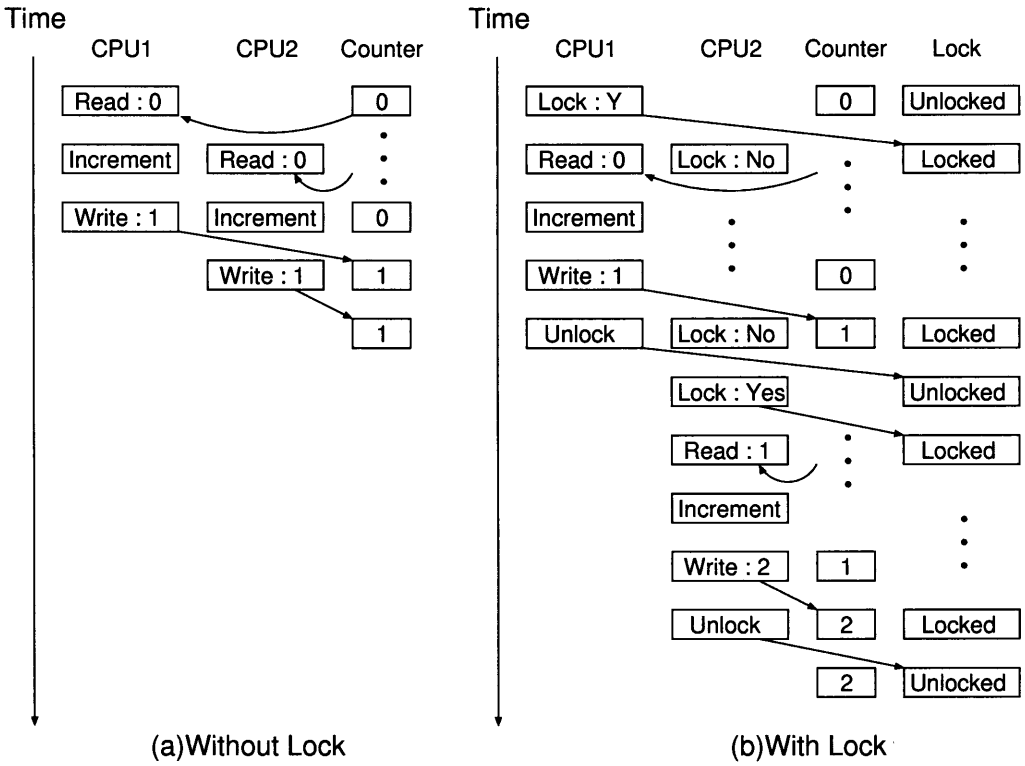


図 2.1: 排他制御の例

険性があり問題となっている.

2.5.2 バリア同期

バリア同期とは，複数のスレッドを1つのグループとして，グループ内の全てのスレッドがある地点に到達するまで実行を停止し，各スレッドの足並みを揃えるための同期手法である．図2.2に，バリア同期を用いる並列処理モデルの一例を示す．この例では，最初に複数のスレッドa～xに処理を分割「Fork」する．次にスレッドa～xのそれぞれが1～nまでの

処理を並列に実行するが，それぞれの段階毎にバリア同期「Barrier」を行って実行タイミングを調整する．そして最後に，全てのスレッドが処理を終えてそれぞれの計算結果を算出し，そのスレッド毎の計算結果を結合「Join」して最終的な結果を得るという流れになる．

一般的なソフトウェアでは，バリア同期を実現するために，バリア変数というものを用いている．具体的には，共有資源であるバリア変数を排他的に操作する．この際，バリア変数は排他制御の説明で述べたカウンタの役目をしており，バリア同期地点に到達したスレッドの数をカウントする．そして，バリア変数がバリアグループ内の全スレッド数に達したかどうかで同期成立を判定し，バリア同期を実現している．

バリア同期は，例えば複数の行列式の和を求めるような，部分解から最終的な解を得たい場合等によく利用される．しかしながらバリア同期は，共有資源であるバリア変数を排他的に扱う同期手法であるため，先に述べた排他制御での問題が同様に問題となっている．

2.6 研究の目的

前節までに，中・細粒度並列処理において，スケジューリングやコンテキスト・スイッチ，そしてスレッド間同期等に起因するオーバーヘッ

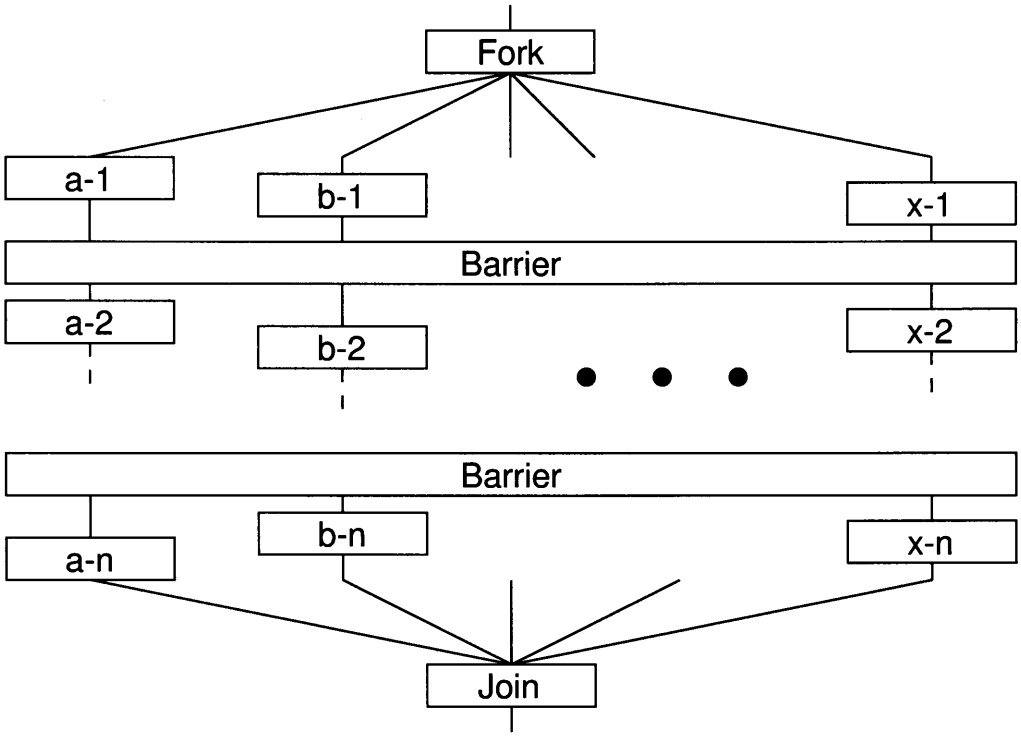


図 2.2: バリア同期モデル

ドが大幅な性能低下を招く危険性がある事を説明した．これらの問題は中・細粒度並列処理を利用する際には必ず対処する必要があるため，関連する研究が広く行われている．その中で，スケジューリングに起因するオーバーヘッドの改善については，佐々木らによりスケジューリング支援ハードウェア (Scheduling Support Hardware; SSH) が提案されている．SSH は，中・細粒度並列処理の効率的な利用を目的としたもので，CPU 上のスレッド実行と並行して専用のハードウェアによりスケジューリングを支援する．また，コンテキスト・スイッチに起因するオーバーヘッド

の改善についても，村松らによりコンテキスト・スイッチ支援システム (Context switch Support System; CSS) が提案されている．CSS は，SSH がスケジューリングに用いる専用のハードウェアにコンテキスト・スイッチ機能を追加実装し，CPU 上のスレッド実行と並行してコンテキスト・スイッチを支援する．SSH と CSS については 4 章にて，より詳しく説明する．

SSH と CSS により，スケジューリングとコンテキスト・スイッチに起因するオーバーヘッドによる性能低下が低減され，逐次処理に対して高性能となる結果も得られている．しかしながら前節で述べた通り，スレッド間同期に起因するオーバーヘッドについても，中・細粒度並列処理においては無視出来ない問題である．また，SSH と CSS を搭載したマルチプロセッサ・アーキテクチャでは，スレッド間同期に起因するオーバーヘッドの対策を行っていないため，依然問題となっている．そこで本研究では，ハードウェアによる同期処理機構を提案して，SSH と CSS を搭載したマルチプロセッサ・アーキテクチャに追加実装する事で，中・細粒度並列処理をより効率的に利用出来る環境の構築を目的とする．

3 関連研究

本章では同期処理の高速化に関する研究についてまとめる．中・細粒度並列処理における同期処理のオーバーヘッドは大きな問題の1つで，これまでにも様々な研究がなされている [11, 12, 13, 14, 15]．本章では，同期処理の高速化に関する研究の中でも代表的な研究をいくつか挙げる．

3.1 Responsive Multithreaded Processor のスレッド間同期機構

村中らは，リアルタイム処理をハードウェアでサポートする Responsive Multithreaded Processor に，ハードウェア同期機構である Sync Unit を搭載する事で高速化を目指す手法を提案している [12]．Sync Unit による同期方式では，全スレッドから書き込める共有レジスタを使用し，そのレジスタにアクセスする専用命令および同期命令を実装している．しかしながら，共有レジスタを用いる方式ではチップ単体での効率は良いが，他のプロセッサとは同期が出来ないため，スケーラビリティに問題がある．

これに対し本研究では，想定するマルチプロセッサ・システムは1チップだけではなく汎用的である．また，排他制御に於いて，CPU 資源の浪費削減を図っている点は共通であるが，本研究では更にロックの獲得を

自動で行うシステムの実装を目指しており，より効率的な排他制御が可能となる．

3.2 笹田らによるSMTプロセッサ用同期機構

笹田らは，Tullsen らが提案したSMT-Block[14] を独自に改良し，ロック獲得予約という効率の良いロック受け渡し方法による高速化を提案している [13]．SMT-Block 方式では lock box というハードウェアをプロセッサ内に用意して，lock box を用いる複雑な専用命令を追加する．笹田らはこの命令の意味を保ったまま，命令の複雑さを解消した新命令を実装した．ロック獲得予約では，ロックを獲得する事を事前に予約する．これにより他のスレッドがロックを解放するとロック獲得予約したスレッドに効率良くロックの受け渡しが行われる．しかしながら，ロック獲得予約には受け渡しが失敗するケースがいくつか存在しており，問題となる場合がある．また，支援する同期処理方式は排他制御のみであり，扱う事の出来るスレッド数についても，ハードウェア量に収まる範囲のみである．

これに対し本研究では，支援する同期処理方式は排他制御とバリア同期を対象としている．また，本研究の同期処理機構は，従来のソフトウェ

アによる同期手法と協調して動作するため，ハードウェア量を意識せずにプログラミング可能である．

4 SSHとCSSを用いた従来のマルチプロセッサ環境

本章では，1章で紹介したスケジューリング支援ハードウェア (Scheduling Support Hardware; SSH) とコンテキスト・スイッチ支援システム (Context switch Support System; CSS) について，より詳細に説明する．

図4.3にSSHとCSSを用いたマルチプロセッサ・アーキテクチャを示す．このマルチプロセッサ・アーキテクチャは，複数のPE(Processor Element)，SSH-m(SSH-master)，共有メモリ (Shared Memory)，メモリ・バス調停器 (Memory Bus Arbiter)，スケジューラ・バス調停器 (Scheduler Memory Arbiter) から成る．メモリ・アーキテクチャとしては，集中あるいは分散共有メモリを想定しており，同一のアドレス空間を共有しているものとする．これは，スレッドを容易に任意のPEに割り当て可能にするためである．

PEは以下の3つのユニットから構成される．すなわち，プログラムの実行を行うCPUと，SSHの一部であるSSH-s(SSH-slave)，そしてCSSで構成されている．SSH-sはCPUとSSH-mのインタフェースを提供する

表 4.1: CPU から見た Register Unit

レジスタ名	物理アドレス
Command	bff10000h
Status	bff10004h
WQ-Status	bff10008h
RQ-Status	bff1000ch
Write Queue 0	bff10100h
Write Queue 1	bff10104h
Write Queue 2	bff10108h
:	:
Write Queue 35	bff1018ch
Read Queue 0	bff10200h
Read Queue 1	bff10204h
Read Queue 2	bff10208h
:	:
Read Queue 35	bff1028ch

もので，CPU の動作と並行して次に実行されるスレッドの取得，新規スレッドのキューイング等を行う．CSS はコンテキスト・スイッチにおけるレジスタの退避／復帰を高速に行うものである．CPU と SSH-s，CSS はオンチップでの実装を想定しており，その内，CPU と SSH-s は 32 ビットのアドレス／データ共有のバスで接続されているものとする．また，CPU と SSH-s の通信はメモリマップド I/O により実現しており，その物理アドレスを表 4.1 に表す．

4.1 SSHの概要

SSHでは，CPU上でのスレッドの実行と並行して，スレッドのスケジューリングを行う．SSHはSSH-mとSSH-sから成っており，スケジューリングは主にSSH-mで行う．SSH-mでは，SSH-sを介して送信された新規スレッドを管理し，スケジューリング・ポリシに従ってスケジュールする．また，SSH-sから要求があった場合，最も優先度の高いスレッドの管理情報をSSH-sに返す．SSH-m及びSSH-sの詳細については，次節以降で詳しく述べる．

SSH-mとSSH-sは専用のスケジューラ・バスにより接続される．スケジューラ・バスの調停は専用の調停器を用いる．このスケジューラ・バス調停器は，固定プライオリティとラウンドロビン型の変動プライオリティを組み合わせたもので，SSH-mからのバス使用要求を最優先とし，それ以外のSSH-sからの要求はラウンドロビンにより決定する．

4.2 SSHの詳細

SSHの構成要素には，スケジューリング等の処理を行うSSH-mと，CPUとSSH-mとのインタフェースを提供するSSH-sがある．また，SSHはサーバ／クライアント・モデルになっており，SSH-mはサーバとして，SSH-s

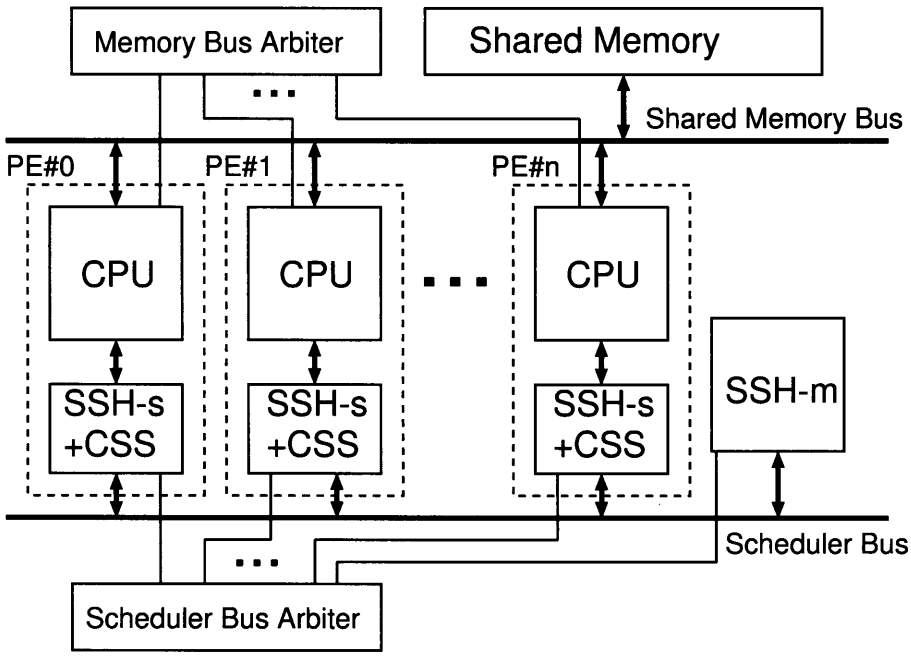


図 4.3: マルチプロセッサ・アーキテクチャ

はクライアントとして機能する．本節では SSH-m 及び SSH-s の詳細なアーキテクチャについて述べる．

4.2.1 SSH-m

図 4.4 に SSH-m のブロック図を示す．SSH-m はスレッドのスケジューリングを行う Hardware Scheduler，及び実行可能キュー等を保持する Global Queue から成る．

各モジュールの機能

各モジュールの機能について，以下に述べる．

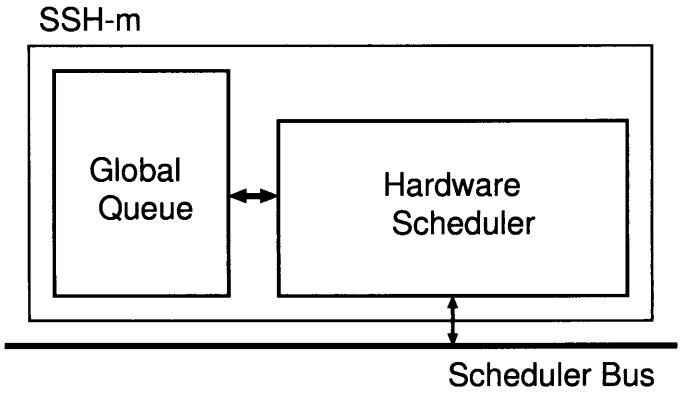


図 4.4: SSH-m のブロック図

Hardware Scheduler: スレッドのスケジューリングを行う．スケジューリング・ポリシとしては，FIFO，ラウンドロビン，Other を実装する．スケジューリング・ポリシの Other とは，Pthread の SCHED_OTHER に相当するものである．

Global Queue: 各スケジューリング・ポリシ毎の実行可能キュー (Ready Queue)，及び待ちキュー (Wait Queue) から成り，SRAM を用いて実装する．

4.2.2 SSH-s

図 4.5 に SSH-s のブロック図を示す．SSH-s は CPU と SSH-m のインタフェースを提供するもので，CPU とのインタフェースを提供する CPU-SSH Interface Unit，スケジューラ・バスを介して SSH-m と通信を行う

SSH-SSH Interface Unit，及びGlobal Queueから先行してロードしたスレッドやGlobal Queueに登録すべきスレッドをバッファリングするためのRegister Unitから成る．

各モジュールの機能

各モジュールの機能について，以下に述べる．

CPU-SSH Interface Unit: CPUの発行したLoad／Store命令のメモリ・アドレスをRegister Unitにマッピングし，内部レジスタの書き込み，読み出しを仲介する．

SSH-SSH Interface Unit: Register Unitを監視し，Read Queueが空になると，SSH-mに要求を出し，次に実行すべきスレッドを取得する．また，Write Queueに有効なデータが入ると，スケジューリングさせるためにSSH-mにスレッドを送信する．

Register Unit: SSH-s及びSSH-mへのコマンドや，次に実行すべきスレッドを格納するRead Queue，新規に生成したスレッド，あるいは，実行権限を譲ったスレッドがSSH-mに送信されるまで保持しておくWrite Queueから成る．

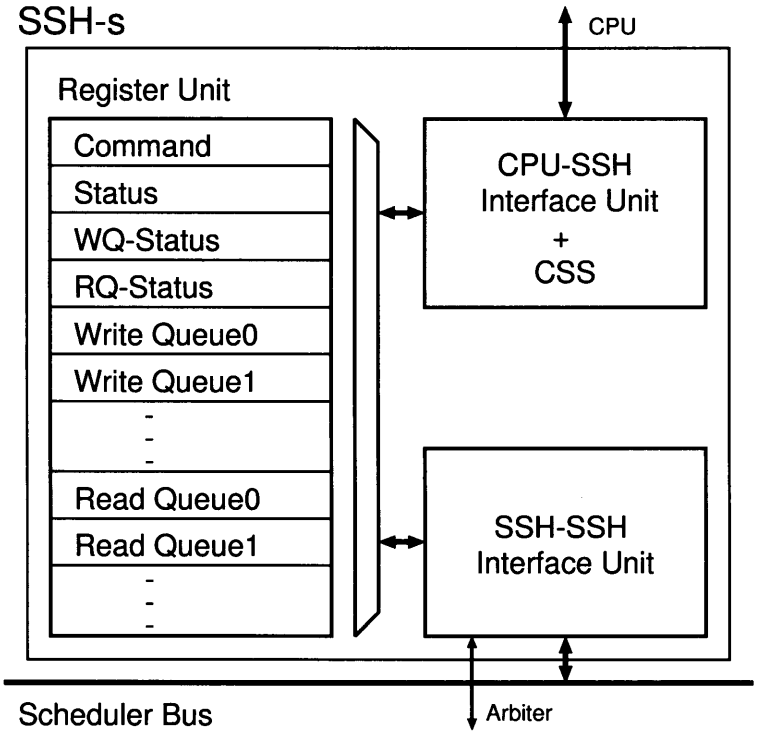


図 4.5: SSH-s のブロック図

Register Unit の詳細

ここでは，Register Unit の詳細について述べる．図 4.5 の Register Unit は 76 本の 32 ビット・レジスタから成る．CPU から各レジスタにアクセスする場合，各レジスタを特定のアドレスにマッピングされた 1 ワードのメモリと見なして，Load／Store 命令を用いてアクセス出来る．各レジスタの機能を以下にまとめる．

Command: SSH-s 及び SSH-m への命令レジスタ．SSH-s に対する命令

は，初期化，停止，タイムスライス設定の3つで，SSH-mに対する命令は，Global Queueの初期化のみである．

Status: Commandレジスタの状態を表すもので，0か1の値をとる．0のときは，命令受け付け状態，1のときは命令実行中を表す．CPUは，Commandレジスタに命令を書く前に，Statusレジスタが0であることを確認する必要がある．また，Commandレジスタに命令を書き込み後，当該レジスタを1にする．

WQ-Status: Write Queueの状態を示すもので，Statusレジスタ同様，0か1の値をとる．Write Queueに未送信のデータが入っている場合は1，空の場合は0である．Statusレジスタ同様，Write Queueにデータを書き込み後，当該レジスタを1にする．

RQ-Status: Read Queueの状態を示すもので，Statusレジスタ同様，0か1の値をとる．Read Queueに有効なデータがある場合は1，空の場合は0である．CPUはRead Queueを読み込み後，当該レジスタを0にする．

Write Queue: 新規に生成されたスレッドや実行権限を譲ったスレッド等，SSH-mに送信し，スケジュール対象となるスレッドを保持する．

Write Queue は Write Queue0-3 までの 4 本で，1 つのスレッドを保持する．Write Queue4-35 までのレジスタは SSH では使用していないが，後述する CSS のために，スレッドのコンテキスト情報を保持出来るように準備している．また，このレジスタは，CPU-SSH Interface Unit を介して CPU により書き込まれ，SSH-SSH Interface Unit に読み出される．

Read Queue: 次に実行すべきスレッドを保持する．Read Queue は，Read Queue0-3 までの 4 本で，1 つのスレッドを保持する．Read Queue4-35 までのレジスタは，Write Queue 同様，SSH では使用していない．また，このレジスタは，SSH-SSH Interface Unit により書き込まれ，CPU-SSH Interface Unit を介して，CPU により読み出される．

4.3 CSS の概要

CSS は SSH で扱うスレッド管理情報にコンテキスト情報を付加する事により，コンテキスト・スイッチ時に伴うレジスタの復帰／退避先を共有メモリではなく，SSH-m にある Global Queue にしている．また，コンテキスト・スイッチ専用のハードウェアにより，CPU 上のスレッド実行と並行してコンテキスト・スイッチを高速に実行する．従来は主記憶で

行っていたコンテキスト情報の管理をスケジューリング・バスを介してSSH-mで行う事により，メモリ・バスの使用頻度が高くなる事による性能低下を回避している．CSSではコンテキスト情報を図4.5にある，Write Queue4-35及びRead Queue4-35を用いて保持している．

4.4 CSSの詳細

図4.6に，SSH及びCSSを搭載したPEのブロック図を示す．CSSではSSH-sが保持しているスレッド情報のうち，コンテキスト情報にあたるRead Queue4-35及びWrite Queue4-35を用いる．SSH-sのWrite Queueが空で，次に実行されるスレッドの情報がRead Queueに保持されている場合，CSSでは現在実行中のスレッドが実行権限を譲る等して，スレッドが切り替わる際に??にあるRegister Switchを制御する事によりコンテキスト・スイッチを高速に行う．これにより，スレッド切り替え時のコンテキスト情報の退避／復元処理に要するサイクル数を削減している．なお，図の簡略化のため，バス・コントローラ，及びCPU内部のメモリ管理ユニット(Memory Managing Unit :MMU)，TLB，乗除算ユニット，システム制御コプロセッサの一部は除いてある．

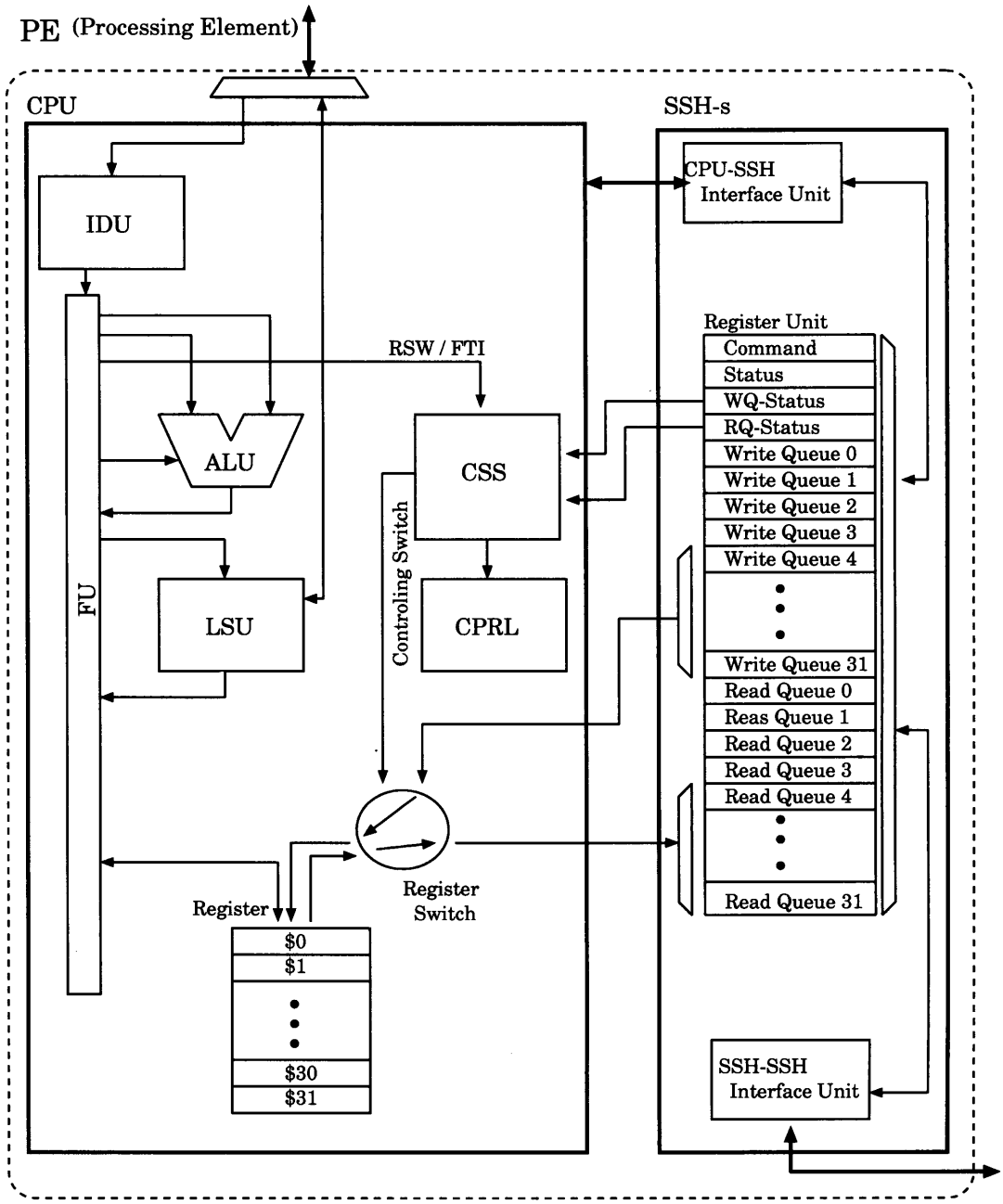


図 4.6: コンテキスト・スイッチ支援システム

4.5 SSH及びCSSの動作

ここで，SSH及びCSSの動作を簡単に説明するために動作の一例を示す．初期条件として，実行開始から十分に時間が経過しており，Global Queueには実行待ちのスレッドが既に存在し，また，当該SSH-sのRead Queue及びWrite Queue共に空であるとする．

STEP1: CPU上ではユーザのスレッドが実行されている．一方，SSH-sはRead Queueが空であるので，スケジューラ・バスの使用要求を出し，バスの使用権を待つ．

STEP2: スケジューラ・バス調停器よりバスの使用権を得る．

STEP3: スケジューラ・バスを介して，SSH-mに次に実行すべきスレッドの情報を要求する．

STEP4: SSH-mは，SSH-sからの要求に従い，Global Queueから最も優先度の高いスレッドの情報を取り出し，SSH-sに返す．

STEP5: SSH-sは次に実行すべきスレッドの情報をSSH-mから受け取り，Read Queueに格納する．

STEP6: CPU上で実行中のスレッドが，実行権限を譲る等して，スレッ

ドの実行が切り替わるタイミングになると，CPU 上ではスレッド・ライブラリへと制御が移る．

STEP7: スレッド・ライブラリにより Write Queue が空であるのを確認すると，それまで実行していたスレッドのコンテキスト情報以外の情報を Write Queue0-3 に書き込む．

STEP8: Write Queue0-3 へ書き込み後，スレッド・ライブラリは Read Queue に次に実行すべきスレッドの情報が書き込まれている事を確認し，Read Queue0-3 の情報を読み出し，CSS にコンテキスト情報の退避／復元を命令する．

STEP9: CSS はスレッド・ライブラリからの命令に従い，それまで実行されていたスレッドのコンテキスト情報を，Write Queue4-35 に書き込み，これから実行されるスレッドのコンテキスト情報を Read Queue4-35 から読み出し，コンテキスト情報の退避／復元を高速に行う．これにより Write Queue に全てのスレッド情報を書き終え，Read Queue は空になる．

STEP10: CPU 上では，スレッド・ライブラリが CSS の動作完了を確認すると，Read Queue から読み出したスレッドの実行を再開する．一

方，SSH-s ではWrite Queue にデータが書き込まれたのを検知し，スケジューラ・バスの使用要求を出し，バスの使用権を待つ．

STEP11: スケジューラ・バス調停器よりバスの使用権を得る．

STEP12: スケジューラ・バスを介して，SSH-m に Write Queue のデータを送信する．

STEP13: SSH-s からスレッドの情報を受け取った SSH-m は，当該スレッドを再スケジューリングし，適切なキューへ追加する．一方，SSH-s は Read Queue が空であることを検知し，再びスケジューラ・バスの使用要求を出し，バスの使用権を得た後，次に実行すべきスレッドの情報を要求する．

SSH 及び CSS を用いたアーキテクチャでは，以上のような動作を繰り返し，高速にスレッドを実行していく．

4.6 SSH 及び CSS の問題点

スケジューリング支援ハードウェア (SSH) を用いたマルチプロセッサ・システムに，コンテキスト・スイッチ支援システム (CSS) を追加する事で，スケジューリング及びコンテキスト・スイッチに起因するオーバー

ヘッドを低減し，高速化を実現した．しかしながら，中・細粒度の並列処理を行うには，同期処理に起因するオーバーヘッドによる性能低下が依然問題となっており，これを解決する必要がある．

具体的には，1度の同期処理に要するオーバーヘッドのサイクル数は100サイクル程度である．それに対して，中・細粒度並列処理では1つのスレッドの生存期間が，100から10000サイクル程度と想定している．そのため，1度の同期処理が1つのスレッドの生存期間に対して，10%から100%にもなる．また，同期手法によっては，同期状態の確認のために何度も同期処理を行う場合がある．その場合，同期が成立していないにも関わらず，同期状態の確認を行った分だけ，他の実行可能スレッドの実行が妨害されて性能が大幅に低下する．特にスレッドの粒度が1000以下の細粒度となると，これが原因で本来見込める性能の数%程度の性能にまで陥る危険性がある．そのため，同期処理の高速化が必要となる．

そこで本研究では，同期処理に起因するオーバーヘッドを低減するために，同期処理支援システム(S3)を，SSH及びCSSに付加し，同期処理についても高速化する事で，より効率的な中・細粒度並列処理環境の実現を目指す．

5 同期処理支援システム (S3) の提案

5.1 S3 の概要

SSH に CSS を付加することにより，スケジューリングに要する時間及びコンテキスト・スイッチに伴うレジスタの退避／復帰処理時間を隠蔽する事で処理の高速化が実現出来る．しかしながら同期処理の回数増加に起因するオーバーヘッドは依然問題として残っており，性能低下を招く原因となっている．そこで，この問題を低減するために，本研究では CPU 上でのスレッドの実行と並行して同期処理を行う事により，同期処理に要する時間を隠蔽する同期処理支援システム (S3) を提案する [16].

5.2 高速化の原理

本節では S3 を実装する事による高速化の原理について述べる．S3 では排他制御とバリア同期を専用ハードウェアにより高速に処理する．本論文ではバリア同期での高速化を例に挙げる．図 5.7 に同じバリアグループであるスレッド T1，T2，T3，T4 と，T1～T4 に全く依存の無いスレッド Other を 2CPU 上で並列に実行している様子を示す．T1～T4 はそれぞれ，前処理 (preprocess)，バリア同期 (barrier)，後処理 (postprocess) という処理を順番に実行する．この内，後処理は T1～T4 の全てのスレッド

がバリア同期を終えた後でのみ，実行出来るものとする．Otherは複数存在し，他スレッドと依存がないものとする．このようなモデルを簡単な具体例で示すと，複数の行列式の和を用いる計算等がある．例えば4つの行列の行列式の和を求めるために，それぞれの行列の行列式を並列に求め，その結果から全ての行列式の和を求める．この場合，それぞれの行列の行列式が求まっていないと，行列式の和を正しく計算出来ないため，バリア同期が必要となる．これを今回のモデルに当てはめると，T1～T4のスレッドはそれぞれの行列の行列式を計算し，その後，他のスレッドが求めた結果を用いて行列式の和を計算する処理を実行する．まず最初に，T1～T4はそれぞれの行列式を並列に計算して求める．この処理が前処理に該当する．次にT1～T4は，他のスレッドが各行列式を求め終えるまで続きを実行出来ないため，他のスレッドが終了するまで待機するための処理をする．これがバリア同期に該当する．そしてT1～T4が，それぞれの行列の行列式を求め終えてバリア同期が成立すると，各スレッドは行列式の和を求めて，それを用いた他の処理等をする．これが後処理に該当する．

図5.7(a)は従来手法であるソフトウェアによる同期処理の流れを示し，図5.7(b)は提案手法であるS3を用いる同期処理の流れを示している．

以下に，従来手法による同期処理と提案手法による同期処理について，それぞれの同期処理の流れを説明する．その後，従来手法と提案手法との相違を述べる．

5.2.1 ソフトウェアによる同期処理

図5.7(a)に示すように，最初にそれぞれのCPUへT1，T3が割り当てられたとする．T1とT3はそれぞれ前処理，バリア同期を終える．続く後処理はT2とT4がまだバリア同期を終えていないので実行出来ず，T1とT3は実行権限を他スレッドへと譲る．

次にCPU1へはT2が割り当てられ，CPU2へはOtherが割り当てられたとする．T2は前処理，バリア同期を終える．しかしT4がまだバリア同期を終えていないので後処理は実行出来ず，T2は実行権限を他スレッドへと譲る．CPU2ではOtherが処理を終えて実行権限を他スレッドへと譲る．

その後，CPU1へはOtherが割り当てられ，CPU2へはT4が割り当てられたとする．CPU2ではT4が前処理，バリア同期を実行する．この時点でT1～T4全てのスレッドがバリア同期を終えたので，T4は続く後処理をそのまま実行する．CPU1ではOtherが処理を終えて実行権限を他スレッドへと譲る．

T4がバリア同期を終えた時点から，T1～T4全てのスレッドがバリア同期を終えた事になる．この事により，残るT1～T3も後処理を実行可能となり，CPUに割り当てられ次第，後処理を実行してそれぞれのスレッドの実行が終了する．ソフトウェアによる同期処理では以上のような流れになる．

5.2.2 S3を用いた同期処理

図5.7(b)に示すように，最初にそれぞれのCPUへT1，T3が割り当てられたとする．T1とT3はそれぞれ前処理を終え，S3にバリア同期を委託して実行権限を他スレッドへと譲る．

次にCPU1へはT2が割り当てられ，CPU2へはOtherが割り当てられたとする．T2は前処理を終えると続くバリア同期をS3に委託して実行権限を他スレッドへ譲る．CPU2ではOtherが処理を終えて実行権限を他スレッドへ譲る．

その後，CPU1へはOtherが割り当てられ，CPU2へはT4が割り当てられたとする．T4は前処理を実行後にS3へバリア同期を委託して実行権限を他スレッドへ譲り，その後CPU2へはOtherが割り当てられる．

一方，S3ではCPU上でのスレッド実行と並行して，T1～T4に委託された時点からそれぞれのバリア同期を実行する．この際，S3で実行する

バリア同期は，ソフトウェアがCPU上で実行する場合に要するサイクル数より，はるかに短いサイクル数で実行出来る．S3が，T1～T4全てのバリア同期を終えると，全てのスレッドが続く後処理を実行可能となる．その後，CPU上でのOtherスレッドが終了してT1～T4はCPUへ割り当てられ次第，それぞれの後処理を終えるとスレッドの実行が終了する．S3を用いた同期処理では以上のような流れになる．

5.2.3 従来手法と提案手法との相違点

図5.7から分かるように，従来手法である(a)に比べ提案手法である(b)の方が，T1～T4全てのスレッドを短時間で終了している．この理由として，バリア同期時間の短縮化が挙げられる．ソフトウェアによる同期処理に比べ，S3ではバリア同期自体を削減している上に，専用のハードウェアにて同期処理を行う事により，CPU上でプログラムが実行するよりはるかに短時間で同様の処理が可能となっているためである．また，T1～T4以外のスレッド，今回の例ではOtherも含めた全体のスループットという点で考えても，従来は同期処理であるバリア同期に充てていたCPU資源を他スレッドが利用する事により効率的になり，スループットが向上している．

以上から，S3を利用する事によりCPUと並行して同期処理を行う事

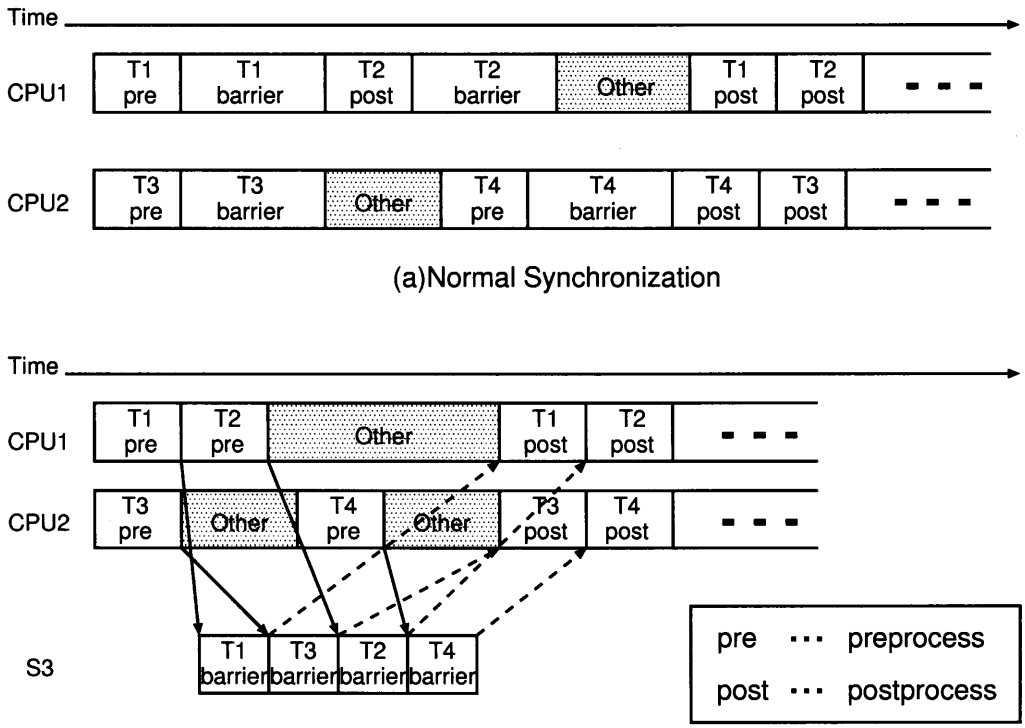


図 5.7: S3 の高速化の原理

で，全体のスループットが改善されるため，効率的な処理が可能になり，処理全体の高速化が実現出来ると考えられる．

5.3 S3 の構成

本節では前節で説明した手法を実現するためのハードウェアについて詳細に説明する．

5.3.1 S3 のブロック図

S3 のブロック図を図 5.8 に示す．S3 は S3 Controller と複数の T_holder，そして Auto CG Unit から成る．

各モジュールの機能

各モジュールの機能ついて，以下に述べる．

S3 Controller: S3 の内の各モジュールを監視して，モジュール間の通信を制御する．また，SSH との通信インタフェースの役割も担う．以下にその動作を簡単に説明する．

まず，SSH から同期処理を必要とするスレッドの情報が送られてくると，複数ある T_holder の中から空き状態の T_holder を選択して，送られてきたスレッド情報をその T_holder へ格納する．次に，T_holder から同期処理の要求があった場合には Auto CG Unit に T_holder の要求を通達し，その T_holder と Auto CG Unit 間の通信を可能にする．そして，T_holder が保持しているスレッドの同期が成立し，そのスレッドが実行可能になると T_holder からスレッド情報を読み出し，SSH へ送信する．

T_holder: S3 の主要モジュールであり，SSH から受け取ったスレッド情報を保持し，同期状態を監視する．

一般的に同期処理は，メモリ上にある同期変数を操作する事により実現している．具体的には，排他制御ではロック変数の獲得や解放という操作，バリア同期ではバリア変数の排他的なインクリメント (又はデクリメント) という操作を行っている．そのため，T_holder では共有メモリ・バスを監視して他スレッドのメモリ・アクセスから同期状態を判定している．また，同期状態によっては同期変数の操作を行うために，共有メモリへのアクセスを Auto CG Unit に要求する．Auto CG Unit を通して同期変数を操作した結果による同期状態の判定も行う．

Auto CG Unit: S3 の共有メモリ・アクセスモジュールである．Auto CG Unit は T_holder からの要求に従い，同期処理の種類に応じた同期変数の操作を行う．同期処理の種類は現在，排他制御とバリア同期をサポートしている．

なお，S3 は図 4.4 にある SSH-m の Hardware Scheduler と，図 4.5 にある SSH-s の SSH-SSH Interface Unit に，それぞれ接続する形で実装する．

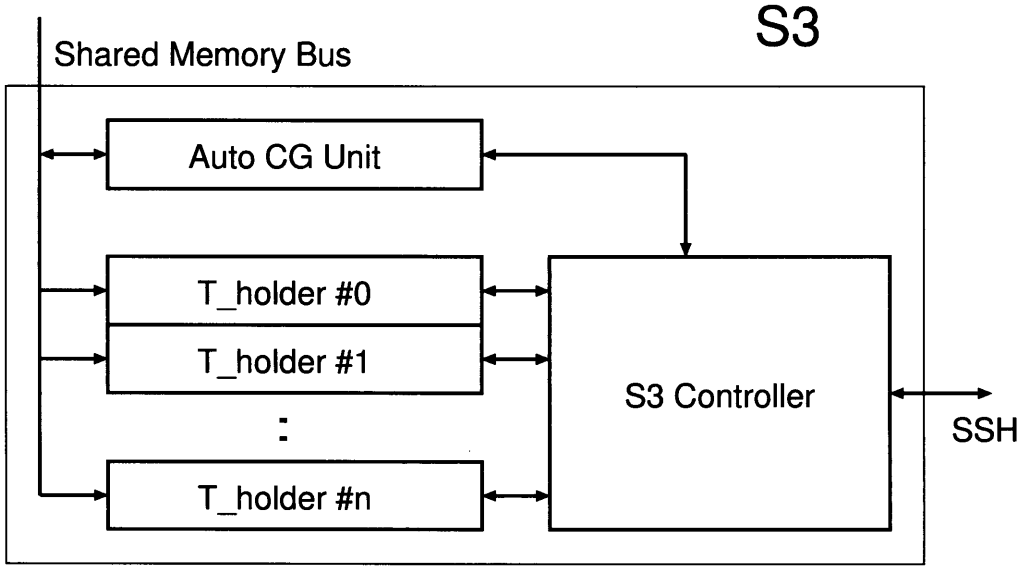


図 5.8: S3 の構成

5.3.2 T_holder のブロック図

次に，S3 に搭載されている主要モジュールである T_holder のブロック図を図 5.9 に示す．T_holder は Selector と Thread Info，Sync Unit から成る．

各モジュールの機能

各モジュールの機能について，以下に述べる．

Selector: SSH から送信されてきたデータを分類して，そのデータの種類に応じて格納先を選択するモジュールである．具体的には，スレッド ID やコンテキスト情報等の基本情報と，同期変数のアドレスや

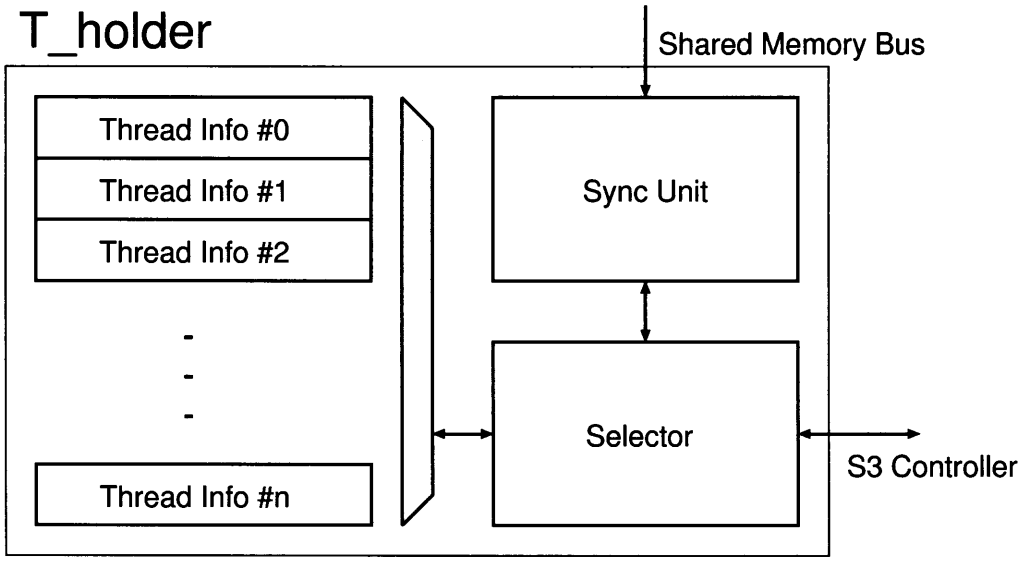


図 5.9: T_holder の構成

同期状態等の同期情報との2種類に分類して，格納先を Thread Info 及び Sync Unit から選択する．

Thread Info: SSH から送信されてきたデータのうち，スレッドの基本情報を格納するキューである．このキューは，図 4.5 の SSH-s にある Write Queue 及び Read Queue と同本数のレジスタ群から成る．

Sync Unit: SSH から送信されてきたデータのうち，同期情報を格納し，その同期情報に応じた処理を行うモジュールである．つまり，同期変数の操作や同期状態の監視等，実際に同期処理を行い，T_holder の動作を制御するのはこのモジュールである．

5.4 S3の動作

ここで，S3の動作を簡単に説明する．ただし，実際に同期処理を行っているのはT_holder内にあるSync Unitであるため，Sync Unitの動作をS3の動作として説明する．

S3は，図5.10のステート・マシンに従って動作する．ステート・マシンは図5.10に示す通り，7種類の状態からなっている．以下にそれぞれの状態について，その状態における動作と共に説明する．

Idle: S3の初期状態で，リセット後はこの状態から動作を開始する．SSHからのスレッド情報を受け付けている状態である．スレッド情報が送信されてくると，スレッド情報の内，同期情報の種類によって状態遷移先が異なる．同期情報の種類が排他制御ならば「Snooping E」へと遷移し，バリア同期ならば「Try Count」へと状態が遷移する．

Snooping E: 排他制御に用いるロック変数を監視している状態である．他スレッドの共有メモリへのアクセスを監視しており，ロック変数が解放された事を検出すると，状態は「Try Lock」へと遷移する．

Try Lock: この状態になると，排他制御に用いられる共有メモリ上のロック変数にアクセスして，ロック獲得を試みる．ロック獲得を試みた

結果が成功であれば「Send Request」へと遷移し，失敗であれば再び「Snooping E」へと状態が遷移する．

Try Count: バリア同期に用いられるバリア変数を，排他的に操作する状態である．この際の排他制御にもロック変数が用いられる．S3 は最初に，このロック変数にアクセスしてロック獲得を試みる．この結果が失敗であれば「Wait」へと状態が遷移し，成功であれば引き続きバリア変数を操作する．バリア変数を操作した後は，排他制御を終えたのでロックを解放する．ロックを解放した後に，バリア同期が成立していたと判断すると「Send Request」へと遷移し，未成立であったと判断すると「Snooping B」へと状態が遷移する．

Wait: バリア変数の操作に失敗した場合の状態である．動作は「Snooping E」と実質的には同等であり，他スレッドのロック変数へのアクセスを監視している．別の状態となっているのは，同期処理の種類が異なり，状態遷移先が異なるためである．ロック解放が検出されれば，再び「Try Count」へと状態が遷移する．

Snooping B: バリア同期に用いるバリア変数を監視している状態である．他スレッドの共有メモリへのアクセスを監視しており，バリア

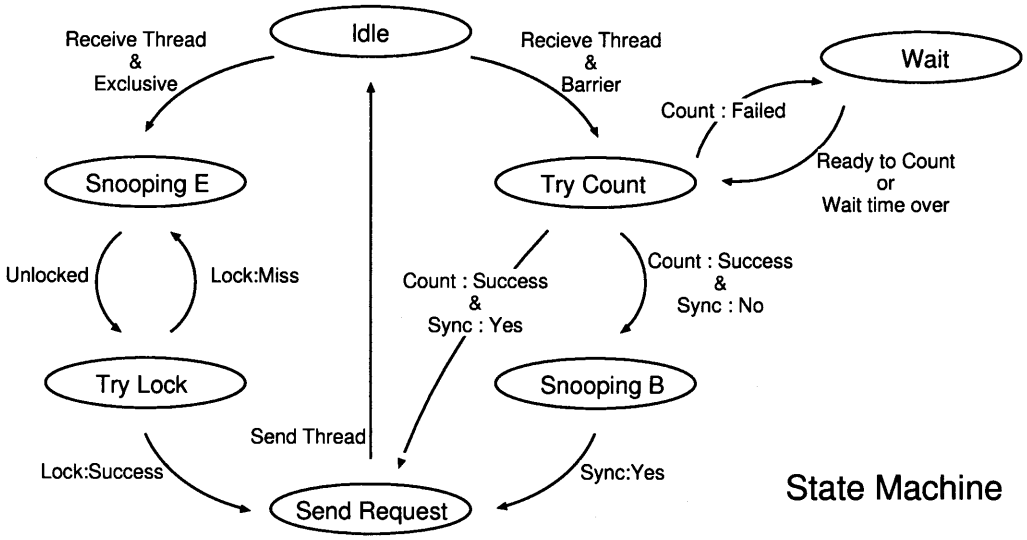


図 5.10: S3 のステート・マシン

同期が成立した事を検出すると,「Send Request」へと状態が遷移する.

Send Request: 保持スレッドが実行可能となったため,スレッド情報の送信を SSH へと要求している状態である.この要求を受けて SSH がスレッド情報を読み出すと,「Idle」へと状態が遷移して,再び SSH からのスレッド情報を受け付けてる状態となる.

6 性能評価

本章では,S3 の有効性を示すために,S3 を含むマルチプロセッサ環境を設計し,性能評価を行う.性能評価では,一般的な並列処理プログラ

ムをモデル化したものを取り上げる．このプログラムを，ソフトウェアのみで実行する場合と従来のSSH+CSSを用いて実行する場合，そして提案手法であるSSH+CSS+S3を用いて実行する場合の3通りの比較評価を行い，S3の有効性を示す．

以降，本稿では従来通りソフトウェアのみで実行する場合を「SOFT」と表し，SSH+CSSを用いて実行する方式を「CSS」，SSH+CSS+S3を用いて実行する方式を「S3」と表す．

6.1 評価環境

評価は，Verilog-HDLシミュレータを用いて行う．シミュレーション評価環境を表6.2に示す．シミュレーション時間を短縮する為，評価はRTLレベルの記述を用いて行う．また，評価プログラムはC言語を用いて作成する．本研究の特徴の一つとして，一般的なスレッド・ライブラリの一つであるPthreadのAPIに準拠しているということがある．その為，本研究で使用した評価用プログラムは，Pthreadを実装している環境があればコンパイル，及び実行することが可能である．また，本研究で用いたCPUコアはMIPS R3000互換の命令セットを実装しているので，C言語を用いて作成した評価用プログラムをクロス開発環境を用いてDELL

表 6.2: シミュレーション評価環境

CPU	Athron 64 3700+
Memory	2GB
Verilog-HDL Simulator	Synopsys Verilog Conpiled Simulator 7.0

表 6.3: 使用したプログラム開発ツール

用途	ツール名	開発元
C Compiler	GCC Ver 3.3.5	GNU
Assembler	GNU assembler Ver 2.15	GNU
Linker	GNU id Ver 2.15	GNU

Optiplex GX260 上でコンパイルし，シミュレーションに利用する．表6.3に使用したクロス環境を示す．

6.2 対象とするマルチプロセッサ環境

マルチプロセッサ環境は，図4.3のものを対象とする．PEは1〜16台までの任意の台数に設定出来る．共有メモリは全PEから等距離にあるUMA(Uniform Memory Access)モデルで，アクセス時間は均一である．Memory Busの調停は，専用のMemory Bus Arbiterで行い，各PEの優先順位はラウンド・ロビンにより動的に決定される．バスの優先順位は全PEとも同一である．Scheduler BusもMemory Busと同様にラウンド・ロビンにより優先順位が決定されるが，SSH-mからの要求には最も高い優先度が割り当てられている．命令メモリは共有メモリに置かれており，

データ・バス，命令・バスはともに32ビットのアドレス／データ分離型である．

6.3 一般的並列処理モデルによる評価

6.3.1 評価方法

S3の性能評価は図6.11に示すような並列処理モデルを用いて，各方式についてのシミュレーション評価を行う．その際用いたパラメータは表6.4の通りである．

図6.11中の大括弧内は，処理に要するサイクル数を示す．プログラムは初期化処理を行った後，スレッドSub_Create()を*CRE_NUM*個生成する．Sub_Create()では更に，スレッドTask()を*THR_NUM* - *CRE_NUM*個生成する．結果として，並列処理を行うスレッドを合計*THR_NUM*個生成する．Pthreadには，スレッドを複数個同時に生成するAPIはない為，ループ文を用いて逐次的にスレッド生成を行う．親スレッドは，生成した順番に子スレッドと結合していき，全スレッドと結合した後，「終了処理」を行ってプログラムを完了する．また，Sub_Create関数及びTask関数の中では並列処理部分を2分割し，スレッド実行中に同期をとる．このようにして，同期処理部分のオーバヘッドを考慮した評価を行う．本研究は，並列処理部分のみの高速化に関する研究ではあるが，逐次処理

部分も含んだ処理全体の性能評価を行う．ここで逐次処理部分とは，すなわち，*INIT* と *TERM* である．この評価プログラムのタスク・グラフを図6.12に示す．

また，バリア同期では同期を行うグループ構成も性能に影響を及ぼすため，スレッドをいくつかのグループに分割する場合の比較を行う．グループ分割は具体的に，全スレッドで1度バリア同期をとる方式を「1G-1S」，2グループに分けて，それぞれのグループ毎に1度バリア同期をとる方式を「2G-2S」，4グループに分けて，それぞれのグループ毎に1度バリア同期をとる方式を「4G-4S」とする．

以上の条件で，次節より性能評価を行う．評価する項目を以下に挙げる．

- スレッドの粒度に対する性能評価
- PE数に対する性能評価 (台数効果)

なお，本評価では処理の実行時間を示す指標としてサイクル数を用いる．これは評価を実機ではなく，Verilog-HDLを用いたシミュレーションにより行っているためである．以降サイクル数と記述した場合には，プログラムの終了までにかかるサイクル数を表す．なお本研究では，マルチプロセッサ環境全体の設計，評価をVerilog-HDLを用いてRTLレベル

int main(){
　　初期化処理 $[INIT]$
　　スレッド Sub_Create() **生成** $[CREATE \times CRE\_NUM]$
　　スレッドの結合 $[JOIN \times CRE\_NUM]$
　　終了処理 $[TERM]$
}

void Sub_Create(){
　　スレッド Task() **生成** $[CREATE \times (THR\_NUM - CRE\_NUM)]$
　　並列処理 $[\frac{THR\_LOAD}{THR\_NUM} \times \frac{1}{2}]$
　　同期
　　並列処理 $[\frac{THR\_LOAD}{THR\_NUM} \times \frac{1}{2}]$
　　スレッドの結合 $[JOIN \times (THR\_NUM - CRE\_NUM)]$
}

void Task(){
　　並列処理 $[\frac{THR\_LOAD}{THR\_NUM} \times \frac{1}{2}]$
　　同期
　　並列処理 $[\frac{THR\_LOAD}{THR\_NUM} \times \frac{1}{2}]$
}

図 6.11: 並列処理モデル

で行っているため，この値を動作周波数で割った値は，実機で動作させた場合の実行時間と等価である．

6.4 スレッドの粒度に対する性能評価

本項では，各方式がどの程度の粒度に適しているかを調べる為，スレッドの粒度に対する評価を行う．この際，PE 数を 8，スレッド数を 128 に

表 6.4: 評価パラメータ

パラメータ	意味
PE	利用できるPE数
CRE_NUM	並列にスレッドを生成するスレッド数
THR_NUM	生成するスレッドの総数
$INIT$	**初期化処理**に要するサイクル数
$CREATE$	スレッド一つ生成するのに要するサイクル数
$JOIN$	スレッド一つと結合するのに要するサイクル数
$TERM$	**終了処理**に要するサイクル数
THR_LOAD	Task() を逐次実行した場合に要するサイクル数 *1

*1 スレッド一つあたりのサイクル数は $\frac{THR\_LOAD}{THR\_NUM}$ となる

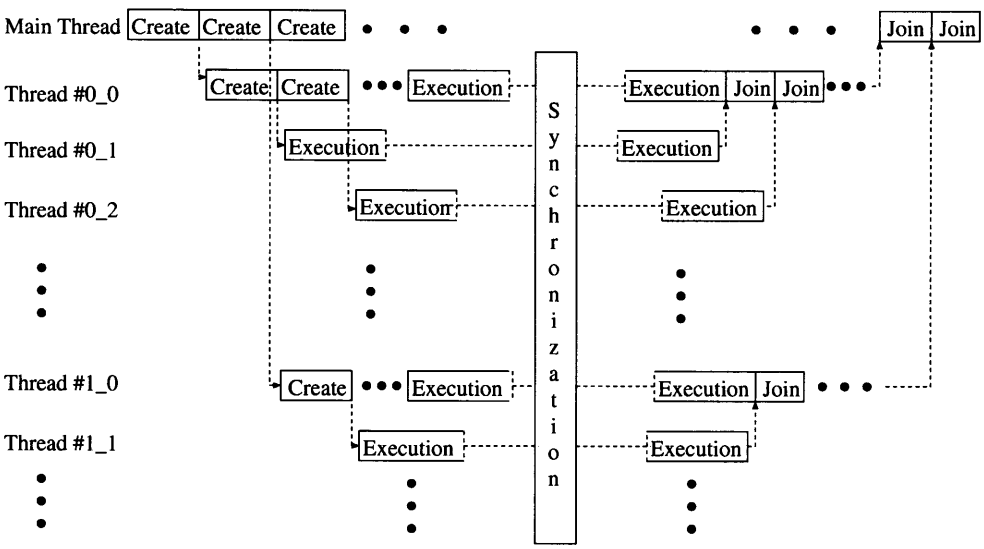


図 6.12: 評価プログラムのタスク・グラフ

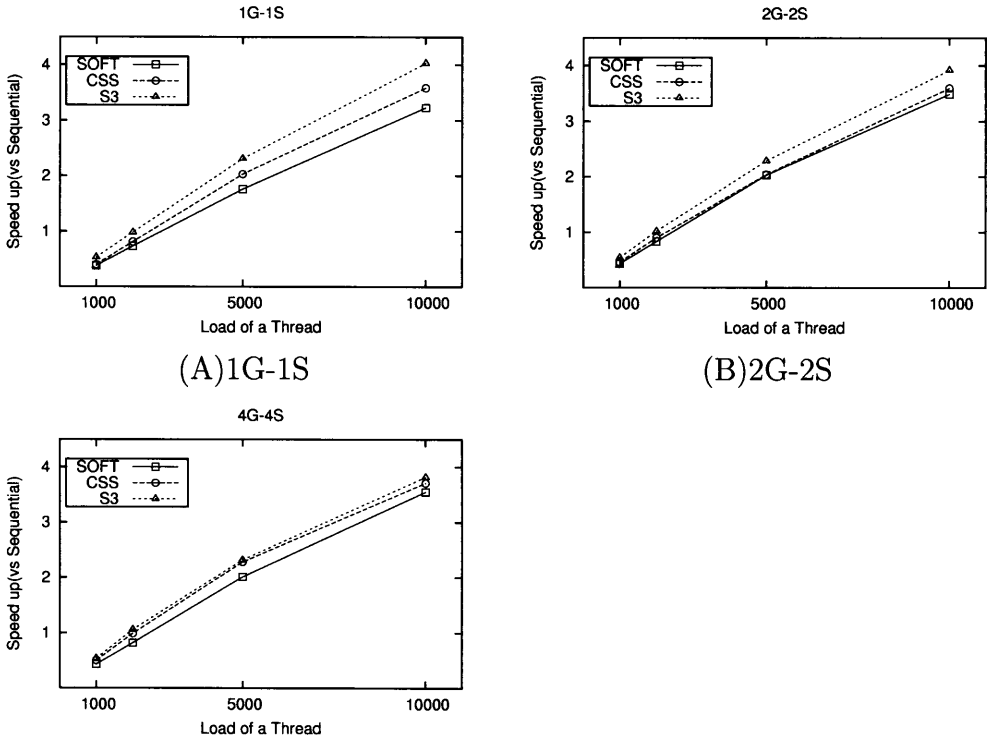
固定した上で，スレッドの粒度を変化させた場合の逐次実行に対する性能を比較する．

6.4.1 評価結果

図6.13に評価結果を示す．図6.13(A)は1G-1S，(B)は2G-2S，(C)は4G-4Sのグループ分割にそれぞれ対応している．

図6.13より，(A)と(B)と(C)の，どのグループ分割条件においても，S3は，従来手法より性能が良くなっているのが分かる．具体的には，S3はスレッドの粒度が2000命令を超えたあたりから逐次処理に対する性能向上が得られている．これに対して，従来手法であるCSSはスレッドの粒度が2500を超えたあたりから，SOFTは3000命令を超えたあたりから，逐次処理に対する性能向上が得られている．つまり，S3を実装する事によって，従来より細粒度での並列処理が可能となり，S3が有効であるといえる．

しかしながら，スレッドの粒度が2000命令より下回ると，全体的に逐次処理に対する性能が悪くなっている．この理由については，次項以降で行うPE数に対する性能評価の結果にて説明するため，ここでは省略する．



(A)1G-1S

(B)2G-2S

(c)4G-4S

図 6.13: スレッド粒度に対する評価 (1G-1S)

6.5 PE数に対する性能評価 (台数効果)

次に本項では，スレッド数を 128 個に固定した上で，プログラムの様々な粒度に於ける PE の台数効果を評価する事で，各方式の比較評価を行う．前項の評価に於いて，逐次処理より同等以上の性能評価結果を得られた，粒度 2000 命令と 5000 命令，そして 10000 命令の場合を評価する．

6.5.1 評価 1:粒度 5000 命令及び 10000 命令における台数効果

粒度 5000 命令及び 10000 命令共に，結果が似た傾向であるため，まとめて示す．図 6.14(A)，(B)，(C) に粒度 5000 命令，及び図 6.15(A)，(B)，(C) に粒度 10000 命令における PE 数に対するそれぞれの手法の評価結果を示す．図 6.14 では (A) が 1G-1S，(B) が 2G-2S，(C) が 4G-4S のグループ分割にそれぞれ対応している．同様に図 6.15 では (A) が 1G-1S，(B) が 2G-2S，(C) が 4G-4S のグループ分割にそれぞれ対応している．

6.5.2 評価 1 の結果

スレッドの粒度 5000 命令及び 10000 命令における台数効果に関しては，全ての手法において，PE 数がいずれの場合でも性能が向上しているのが分かる．この内，S3 に関してはスレッドの粒度が 5000 命令の場合に逐次処理に対して性能が最大 140%倍向上している．スレッドの粒度が 10000 命令の場合は逐次処理に対して性能が最大 320%倍向上している．

また，従来手法である SOFT や CSS に対しても常に同等以上の性能が得られている．具体的には，スレッドの粒度が 5000 命令の場合，SOFT に対しては最大約 39%，CSS に対しては最大 53%性能が向上している．スレッドの粒度が 10000 命令の場合は，SOFT に対して最大約 28%，CSS

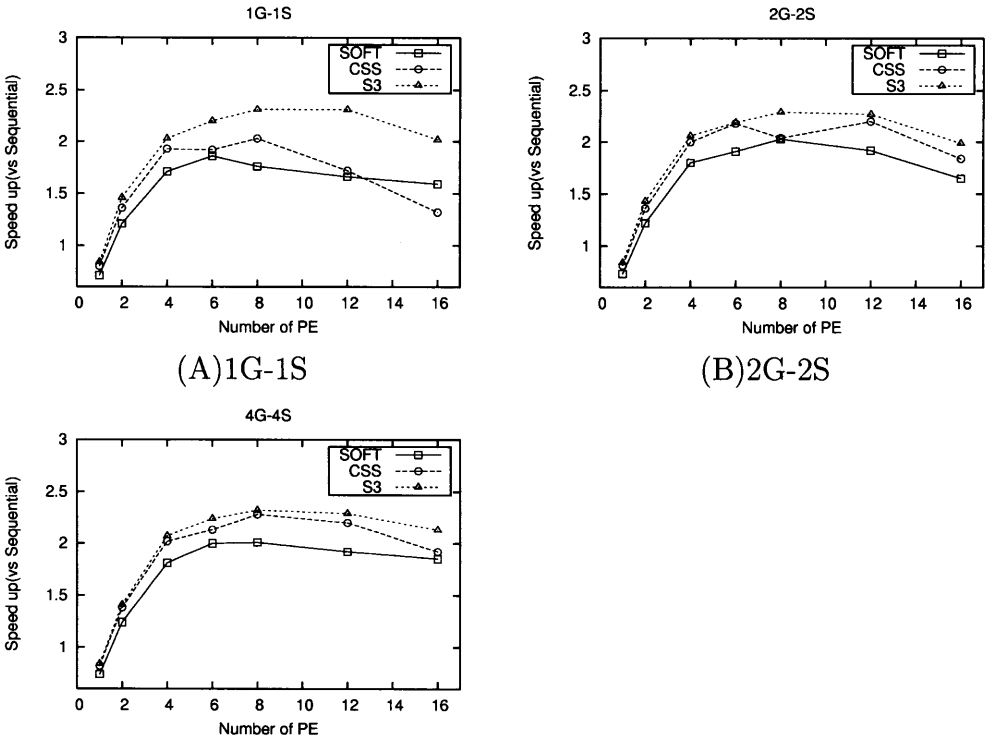


図 6.14: PE 数に対する評価 (スレッド数 128 粒度 5000 命令)

に対しては最大 39%性能が向上している．これにより，スレッドの粒度 5000 命令及び 10000 命令における台数効果に関して，S3 の有効性を示せたと言える．

6.6 評価 2:粒度 2000 命令における台数効果

図 6.16(A)，(B)，(C) に，粒度 2000 命令における PE 数に対するそれぞれの手法の評価結果を示す．図 6.16(A) は 1G-1S，(B) は 2G-2S，(C) は 4G-4S のグループ分割にそれぞれ対応している．

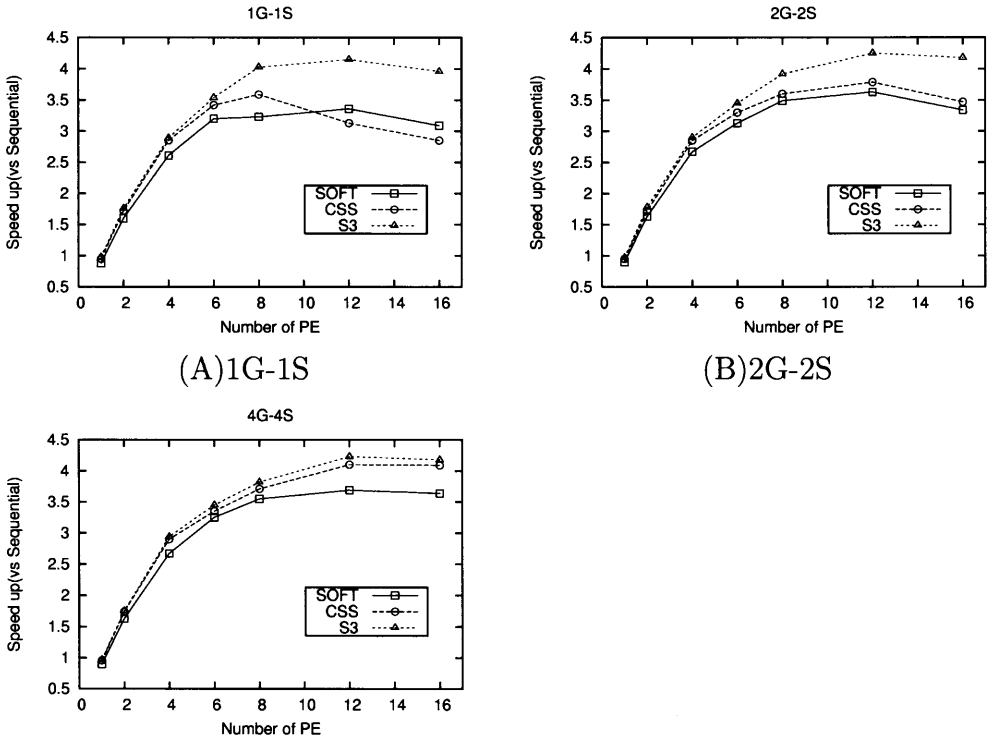


図 6.15: PE 数に対する評価 (スレッド数 128 粒度 10000 命令)

6.6.1 評価 2 の結果

以上より，スレッドの粒度2000命令における台数効果に関しては，PE数が4台から8台の場合でS3とCSSのみ，逐次処理と同等程度の性能を得ている．しかしながら，全体的には逐次処理より性能が悪くなっているのが分かる．この理由として，*CREATE*と*JOIN*に起因するオーバーヘッドが挙げられる．*CREATE*と*JOIN*は，排他的な処理を必要とする共有メモリ・アクセスを，何度も行なっている．その事により，*CREATE*

と *JOIN* に関しては，並列化を行っても効果があまり得られず，性能向上が難しくなる．また，中・細粒度並列処理では，スレッド数が増加し，粒度が小さくなる．これに伴い，*CREATE* と *JOIN* に起因するオーバーヘッドが，本質的な処理である *THR_LOAD* と比較して，大きくなる．そのため，並列化の効果が充分に得られない状況となる．

これを説明するために，図6.17に逐次処理と並列処理の実行結果から，それぞれの処理の内訳を示し，並列処理化の効果をグラフにして表す．これは，スレッド数が128，粒度が2000命令，グループ分割が1G-1Sの場合の，逐次処理プログラムと並列処理プログラムの実行結果から得られた処理の大凡の内訳である．

図6.17の上段は逐次処理，中段はPE数が1台のときの並列処理，下段はPE数が8台のときの並列処理をそれぞれ表す．この条件の下で並列化した場合，図を見て分かる通り，並列処理化したプログラムでは，*CREATE* と *JOIN* が処理全体の50%以上を占めている．そのため，PE数を増やして並列に処理を実行しても，並列化の効果を得られる部分が少なくなり，性能向上が難しくなっている．この結果，図6.16(A) と (B) と (C) のように，スレッドの粒度が2000命令の場合，逐次処理に対して並列処理は，全体的に性能が良くない評価結果になったと考えられる．

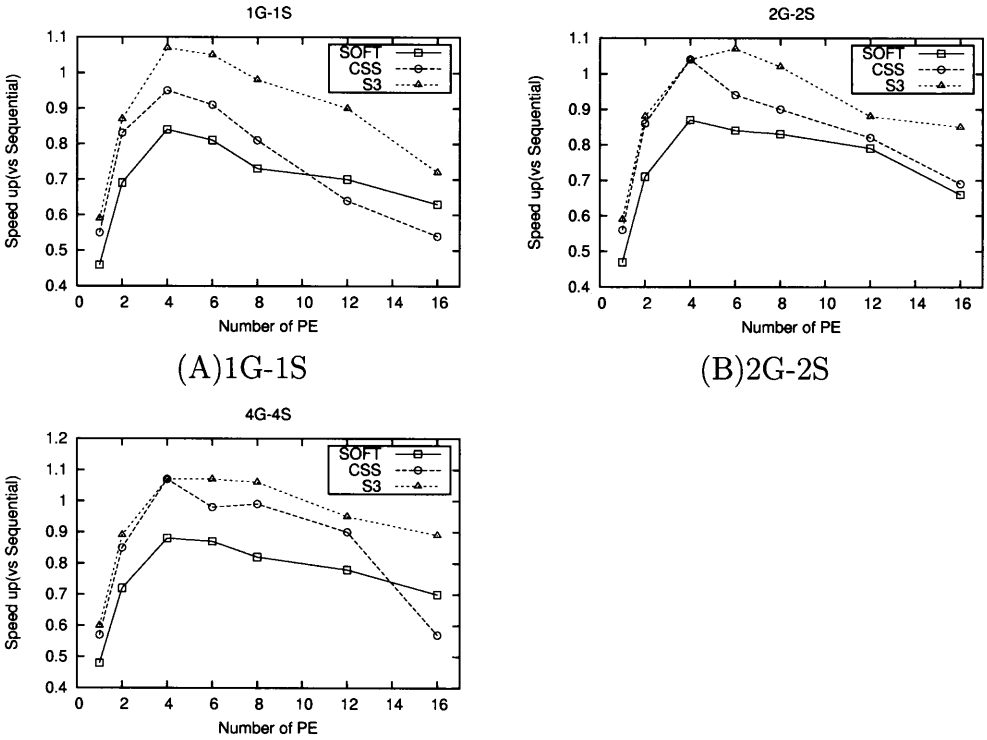


図 6.16: PE数に対する評価 (スレッド数 128 粒度 2000 命令)

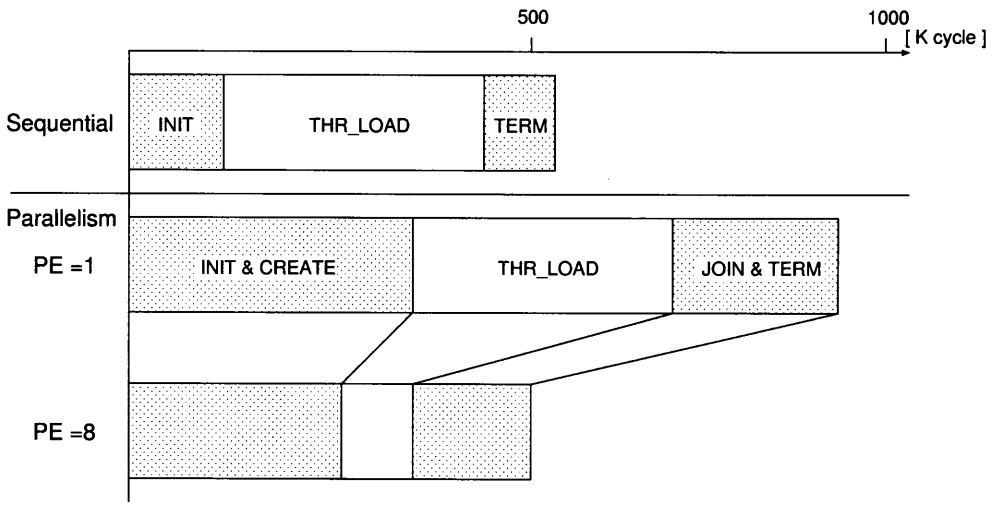


図 6.17: 各処理の内訳 (スレッド数 128 粒度 2000 命令)

6.7 考察

本章では，スレッドの粒度に対する性能評価と，PE数に対する性能評価をそれぞれ行った．その結果から，提案手法であるS3が従来手法であるCSSとSOFTより性能が良くなる事が分かった．

具体的には，PE数に対する性能評価において，S3を用いた提案方式では，従来方式であるCSSより最大39%，SOFTよりは最大53%の性能向上を得る事が出来た．また，スレッドの粒度に対する性能評価においても，S3では2000命令以上の粒度で逐次処理と同等以上の性能を得る事が出来た．これに対して，従来手法であるCSSは約2500，SOFTでは約3000命令以上の粒度で逐次処理と同等以上の性能になっていた．この事から，S3の方がCSSやSOFTと比較して，より細粒度で性能向上が得られる事を示せた．

以上より，本章での性能評価において，従来手法であるCSSやSOFTに対して，提案手法であるS3の有効性を示す事が出来た．

7 結論と今後の展望

本論文では，スケジューリング支援ハードウェア(SSH)及び，コンテキスト・スイッチ支援システム(CSS)を用いたマルチプロセッサ環境に，

同期処理支援システム (S3) を追加することで，中・細粒度並列処理を有効に利用可能なシステムの構築を行った．

6章で行った評価によると，PE数に対する性能評価において，S3を用いた提案方式では，従来方式であるSOFTより最大53%，CSSより最大39%の性能向上を得る事が出来た．また，スレッドの粒度に対する性能評価においても，S3では2000命令以上の粒度で逐次処理より性能向上を得る事が出来た．これに対して，従来手法であるCSSは約2500，SOFTでは約3000命令以上の粒度で逐次処理より性能向上を得ていた．この事から，S3の方がCSSやSOFTと比較して，より細粒度で性能向上が得られる事を示せた．

以上より，本研究で提案するハードウェアによる同期処理支援システム (S3) が，従来手法であるSOFT及びCSSより有効である事を示せた．しかしながら，中・細粒度並列処理を行うには，まだ改良の余地がある．そこで，今後の展望として，次のような項目が考えられる．

S3が行う同期処理の更なる高速化: S3を実装する事により，CPU上での同期処理に要するサイクル数を削減出来た．しかしながら，S3がCPU上のスレッド実行と並行して行っている同期処理は，サイクル数を要しており，これを削減出来ると考えている．具体的には現

在，S3を利用するためのスレッド情報は，CPUからSSHを介して送信されている．そのため，CPU上でスレッド情報の送信を開始してからS3に到達するまでの間に，余分な時間がかかっている．これを解決するために，CPUから直接S3にスレッド情報を送信出来るようにする．

CREATEとJOINの処理時間短縮: 6章の性能評価結果より，スレッドの粒度が2000命令以下において，従来手法と提案手法とも，逐次処理に対して性能が悪くなっていた．この原因の1つとして，スレッドの生成*CREATE*と結合*JOIN*に起因するオーバーヘッドがあると述べた．中・細粒度並列処理をより効率的に用いるには，これらの処理の高速化が必要となる．この問題を低減するために，ライブラリを改良する，又はスレッドの生成と結合を支援するハードウェアを実装する等の手段がある．

現在，上記のうち，S3が行う同期処理の更なる高速化を実装中である．

謝辞

本研究の機会を与えて頂き，ご指導頂いた近藤利夫教授に深甚なる謝意を表します．コンピュータ・アーキテクチャという非常に興味深い分野

へ導いて頂き，また常にご指導頂いた大野和彦講師，佐々木敬泰助手に深く感謝致します．

参考文献

[1] 佐々木 敬泰, 西村 直己, 弘中 哲夫, 吉田 典可: マルチプロセッサ用スケジューリング支援ハードウェアの提案とシミュレーション評価, 電子情報通信学会論文誌, Vol.J84-D-I, No.11, pp.1515–1531 (2001).

[2] Naoki Nishimura, Takahiro Sasaki and Tetsuo Hironaka: "Prototype Microprocessor LSI with Scheduling Support Hardware for Operating System on Multiprocessor System," Asia and South Pacific Design Automation Conference 2000 (ASP-DAC2000),pp.29–30 (2000).

[3] 西村 直己，佐々木 敬泰，木山真人，弘中 哲夫，藤野 清次: マルチプロセッサ・システムに於けるスケジューリング支援ハードウェア，信学技報 CPSY99-57，pp.79-86(1999).

[4] Tullsen, D.M., Eggers, S.and Levy,H.M.: Simultaneous Multithreading: Maximizing On-Chip Parallelism, Proceedings of the 22th Annual International Symposium on Computer Society, pp.294-403(1995).

[5] 田端邦男，田浦健次朗，米沢明憲:C プログラムにおける Lazy Task Creation，Vol.1997 No.78，pp.79-84(1997).

[6] 高田正法，入江英嗣，服部直也，渡邊翔太，清水一人，坂井修一: クラスタ型プロセッサにおける SMT 実行，情報処理学会研究報告 Vol.2005 No.19，pp.37-47(2005).

[7] 石坂一久，中野啓史，八木哲志，小幡元樹，笠原博徳: 共有メモリマルチプロセッサ上でのキャッシュ最適化を考慮した粗粒度並列タスク並列化，情報処理学会論文誌 Vol.43 No.4, pp.958-970(2002).

[8] 小幡元樹，石坂一久，神長浩気，中野啓史，吉田明正，笠原博徳: 商用 SMP 上での粗粒度タスク並列処理，情報処理学会研究報告 Vol.2002 No.9，pp.55-60(2002).

[9] 内田貴之，木村啓二，小高剛，笠原博徳，シングルチップマルチプロセッサにおけるマルチグレイン並列処理，情報処理学会研究報告 Vol.2002 No.9，pp.13-18(2002).

[10] 木村啓二，加藤孝幸，笠原博徳:近細粒度並列処理用シングルチップマルチプロセッサにおけるプロセッサコアの評価，情報処理学会論文誌 Vol.42 No.4，pp.1234-1245(2001).

[11] 山脇彰，岩根雅彦:共有変数の同期を考慮したキャッシュ構成とその予備実験，情報処理学会研究報告 Vol.2002，No.81，pp.133-138(2002).

[12] 村中延之，伊藤務，新井誠一，山崎信行:Responsive Multithread Processor のスレッド間同期機構の設計と実装，情報処理学会研究報告 2005-SLDM-119, pp.121-126(2005).

[13] 笹田耕一，佐藤未来子，内倉要，加藤義人，大和仁典，中条拓伯，並木美太郎: SMT プロセッサにおける同期方式の検討，情報処理学会研究報告 2005-ARC-162, pp.31-36(2005).

[14] Tullsen,D.M,Lo,J.L.,Eggers,S.J. and Levy,H.M.: Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor, HPCA '99: Proceedings of the Fifth International Symposium on High Performance Computer Architecture, IEEE Computer Society, pp.54-58(1999).

[15] 尾形航，吉田明正，合田憲人，岡本雅巳，笠原博徳: スタティックスケジューリングを用いたマルチプロセッサシステム上での無同期近細粒度並列処理，情報処理学会論文誌 Vol.35 No.4，pp.522-530(1994).

[16] 伊賀崇幸，佐々木敬泰，大野和彦，近藤利夫: 中粒度並列処理用ハードウェア同期機構の提案，情報処理学会研究報告 2006-ARC-170，pp.79-84(2006).